# BINARY SEARCH TREE

DECEMBER 5, 2020

# Table of Contents


Binary Search Tree (Background): 2

Introduction to Binary Search Trees 2

- Properties 2
- Searching a Key 2
- Insertion of Key 3
- Deletion of Key 4

Problems 8

1. Print Common Nodes in two BSTs 8
2. Lowest Common Ancestor in a BST 10
3. Print BST elements in given range 12
4. Pair Sum in BST 12
5. Smaller on Right 13
6. Floor in BST 14
7. Ceil in BST 15
8. Find kth smallest in BST 16
9. Preorder to postorder 19
10. Merge two BST 's 20
11. Fixing Two nodes of a BST 22

Self-Balancing BST 23

- How do Self-Balancing BSTs Balance? 23
- Types of Self-Balancing BSTs 24
- AVL Trees 24
  - Rotations in AVL Trees 25
  - Insertion examples 26
- Red-Black Tree 29
  - Introduction 29
  - Rules for Red-Black Tree 29
  - Why Red-Black Trees? 29
  - Comparison with AVL 29
  - Black Height of a Red-Black Tree 29
- Applications of Self-Balancing Binary Search Trees 30

Problems 30

1. Check for BST 30
2. Find closest element in BST 31

3. Convert level-order traversal to BST ..... 31
4. Count BST nodes that lie in the given range ..... 33
TreeSet and TreeMap in Java ..... 34
Problems on TreeSet/TreeMap ..... 35
1. Vertical Traversal of binary tree ..... 35
2. Top view of Binary Tree ..... 37
3. Bottom view of Binary Tree ..... 39
4. Ceiling on left side in an Array ..... 40
5. Vertical sum ..... 41

## Binary Search Tree (Background):

| | Array (unsorted) | Array (sorted) | Linked List (unsorted) | Linked List (sorted) | BST (Balanced) | Hash Table |
|---|---|---|---|---|---|---|
| **Search** | O(n) | O(logn) | O(n) | O(n) | O(logn) | O(1) |
| **Insert** | O(1) | O(n) | O(1) | O(n) | O(logn) | O(1) |
| **Delete** | O(n) | O(n) | O(n) | O(n) | O(logn) | O(1) |
| **Find closest** | O(n) | O(logn) | O(n) | O(n) | O(logn) | O(n) |
| **Sorted traversal** | O(nlogn) | O(n) | O(nlogn) | O(n) | O(n) | O(nlogn) |

Balanced BSTs does all the operations in $O(log\,n)$ time. If BST is not balanced it requires O(height of BST) time on average (consider left skewed BST).

## Introduction to Binary Search Trees

### Properties

- The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.
- In-order traversal of BST gives sorted list.

The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast in comparison to normal Binary Trees. If there is no ordering, then we may have to compare every key to search a given key.

### Searching a Key

Using the property of Binary Search Tree, we can search for an element in O(h) time complexity where h is the height of the given BST.

```c
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // Key is greater than root's key
    if (root->key < key)
       return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

## Insertion of Key

Inserting a new node in the Binary Search Tree is always done at the leaf nodes to maintain the order of nodes in the Tree. The idea is to start searching the given node to be inserted from the root node till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

**Solution 1 (Iterative)**

```java
    Node insert(Node root, int key)
    {
        // your code here
        Node new_node = new Node(key);
        if(root==null)
            return new_node;
        Node ptr = root, prev = null;
        while(ptr!=null){
            prev = ptr;
            if(ptr.data>key)
                ptr = ptr.left;
            else
                ptr = ptr.right;
        }
        if(prev.data>key)
            prev.left = new_node;
        else if(prev.data<key)
            prev.right = new_node;
        return root;
    }
```

**Solution 2 (Recursive)**

```java
Node insert(Node root, int Key)
{
    // your code here
    if(root==null){
        root = new Node(Key);
        return root;
    }
    if(root.data>Key)
        root.left = insert(root.left, Key);
    else if(root.data<Key)
        root.right = insert(root.right, Key);
    return root;
}
```

**Time Complexity:** The worst-case time complexity of search and insert operations is O(h) where **h**, is height of Binary Search Tree. In the worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

## Deletion of Key

The task is to search that node in the given BST and delete it from the BST if it is present.

When we delete a node, three cases may arise:

1. **Node to be deleted is leaf:** Simply remove from the tree. *(Super-simple)*
2. **Node to be deleted has only one child:** Copy the child to the node and delete the child. (Simple)

```
    50                            50
 /      \         delete(30)     /   \
30       70       --------->   40     70
  \     /  \                         /  \
   40  60   80                      60   80
```

3. **Node to be deleted has two children:** We have two choices here either we use in-order successor or in-order predecessor of node to be deleted.
   Find in-order successor of the node. Copy contents of the in-order successor to the node and delete the in-order successor. Note that in-order predecessor can also be used.

```
    50                            60
 /      \         delete(50)     /   \
40       70       --------->   40     70
        /  \                            \
       60   80                           80
```

**Solution 1 (Recursive)**
**Using in-order successor**

1. Recursively find the node that has the same value as the key, while setting the left/right nodes equal to the returned subtree
2. Once the node is found, have to handle the below 4 cases
   a. node doesn't have left or right - return null
   b. node only has left subtree- return the left subtree
   c. node only has right subtree- return the right subtree
   d. node has both left and right - find the minimum value in the right subtree, set that value to the currently found node, then recursively delete the minimum value in the right subtree

```java
public TreeNode deleteNode(TreeNode root, int key) {
    if(root==null)
        return root;
    if(root.val>key)
        root.left = deleteNode(root.left, key);
    else if(root.val<key)
        root.right = deleteNode(root.right, key);
    else{
        if(root.left==null)
            return root.right;
        if(root.right==null)
            return root.left;
        TreeNode inorder_suc = getInOrderSuccessor(root.right);
        root.val = inorder_suc.val;
        root.right = deleteNode(root.right, inorder_suc.val);
    }
    return root;
}
TreeNode getInOrderSuccessor(TreeNode root){
    if(root.left==null)
        return root;
    return getInOrderSuccessor(root.left);
}
```

**Using in-order predecessor**

```
    public TreeNode deleteNode(TreeNode root, int key) {
        if(root==null)
            return root;
        if(root.val>key)
            root.left = deleteNode(root.left, key);
        else if(root.val<key)
            root.right = deleteNode(root.right, key);
        else{
            if(root.left==null)
                return root.right;
            if(root.right==null)
                return root.left;
            TreeNode inorder_pred = getInOrderPredeccessor(root.left);
            root.val = inorder_pred.val;
            root.left = deleteNode(root.left, inorder_pred.val);
        }
        return root;
    }
    TreeNode getInOrderPredeccessor(TreeNode root){
        if(root.right==null)
            return root;
        return getInOrderPredeccessor(root.right);
    }
```

**Solution 2 (Iterative)**

1. Find node with value X [logn time] keep track of parent
2. If node is not present in given root return root
3. Else, if node.right exists then append the node.left (left subtree) to the leftmost node of node.right (right subtree)
4. If node to be deleted is not root itself
    a. If node.right exist return root.right
    b. Else return root.left
5. Else
    a. If node is left child of parent then parent.left = ptr.right
    b. Else parent.right = ptr.right

```java
    public static Node deleteNode(Node root, int X)
    {
        // code here.
        Node ptr=root, parent = null;
        while(ptr!=null && ptr.data!=X){
            parent = ptr;
            if(ptr.data>X)
                ptr = ptr.left;
            else
                ptr = ptr.right;
        }
        // if X is not found
        if(ptr==null)
            return root;
        // if root itself is to be deleted
        if(ptr.right!=null){
            Node left_most = getLeftMost(ptr.right);
            left_most.left = ptr.left;
        }
        if(parent==null){
            if(root.right!=null)
                root = root.right;
            else
                root = root.left;
        }
        // if node to be deleted is found
        else {
            if(parent.data>X)
                parent.left = ptr.right;
            else
                parent.right = ptr.right;
        }
        return root;
    }
    static Node getLeftMost(Node root){
        if(root.left==null)
            return root;
        return getLeftMost(root.left);
    }
```

## Problems

### 1. Print Common Nodes in two BSTs

Given two Binary Search Trees (without duplicates). Find need to print the common nodes in them. In other words, find intersection of two BSTs.

```
Input:
BST1:
                  5
               /     \
             1         10
           /   \      /
          0     4    7
                      \
                       9
BST2:
                10
              /    \
             7      20
           /   \
          4     9
Output: 4 7 9 10
```

**Solution 1 (Uses extra space)**

i. Do in-order traversal of both the trees. Store one traversal in ArrayList and other in HashSet (this will do time complexity optimization).
ii. Traverse ArrayList and check this element present in HashSet or not. If yes add it to res otherwise not.

```java
    public static ArrayList<Integer> printCommon(Node root1,Node root2)
    {
        //add code here.
        ArrayList<Integer> tree1 = new ArrayList<>();
        HashSet<Integer> tree2 = new HashSet<>();
        Stack<Node> st = new Stack<>();
        while(!st.isEmpty() || root1!=null){
            if(root1!=null){
                st.push(root1);
                root1 = root1.left;
            }
            else{
                root1 = st.pop();
                tree1.add(root1.data);
                root1 = root1.right;
            }
        }
        while(!st.isEmpty() || root2!=null){
            if(root2!=null){
                st.push(root2);
                root2 = root2.left;
            }
            else{
                root2 = st.pop();
                tree2.add(root2.data);
                root2 = root2.right;
            }
        }
        ArrayList<Integer> res = new ArrayList<>();
        for(int i : tree1){
            if(tree2.contains(i))
                res.add(i);
        }
        return res;
    }
```

**[Better] Solution 2 (Uses only O(height pf BST) space)**

The idea is to use iterative inorder traversal. We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element during the inorder traversal, we print it. Else, if the elements are not same, we should accordingly go to right of first or second tree. Also, when you go for the right subtree if elements are not equal, then you should keep track of node of another subtree.

```java
    public static ArrayList<Integer> printCommon(Node root1,Node root2)
    {
        //add code here.
        Stack<Node> st1 = new Stack<>();
        Stack<Node> st2 = new Stack<>();
        ArrayList<Integer> res = new ArrayList<Integer>();
        while(true){
            if(root1!=null){
                st1.push(root1);
                root1 = root1.left;
            }
            else if(root2!=null){
                st2.push(root2);
                root2 = root2.left;
            }
            else if(!st1.isEmpty() && !st2.isEmpty()){
                root1 = st1.peek();
                root2 = st2.peek();
                if(root1.data == root2.data){
                    res.add(root1.data);
                    root1 = root1.right;
                    root2 = root2.right;
                    st1.pop();
                    st2.pop();
                }
                else if(root1.data<root2.data){
                    root1 = root1.right;
                    st1.pop();
                    root2 = null;
                }
                else{
                    root2 = root2.right;
                    st2.pop();
                    root1 = null;
                }
            }
            else
                break;
        }
        return res;
```

## 2. Lowest Common Ancestor in a BST

Given a Binary Search Tree (with all values unique) and two node values. Find the Lowest Common Ancestors of the two nodes in the BST.

```
Input:
              5
           /     \
         4         6
        /            \
       3               7
                        \
                          8
n1 = 7, n2 = 8
Output: 7
```

```
class BST
{
    // Returns the LCA of the nodes with values n1 and n2
    // in the BST rooted at 'root'
    Node LCA(Node root, int n1, int n2)
    {
        // code here.
        if(root==null)
            return root;
        if(root.data==n1 || root.data==n2)
            return root;
        if(root.data>n1 && root.data>n2)
            return LCA(root.left, n1, n2);
        if(root.data<n1 && root.data<n2)
            return LCA(root.right, n1, n2);
        return root;
    }

}
```

## 3. Print BST elements in given range

Given a Binary Search Tree and a range. Find all the numbers in the BST that lie in the given range.

**Note**: Element greater than or equal to root go to the right side.

```
Input:
       17
     /    \
    4      18
  /   \
 2     9
l = 4, h = 24
Output: 4 9 17 18
```

```
class Solution
{
    public static ArrayList<Integer> printNearNodes(Node root, int low, int high)
    {
      // code here.
      ArrayList<Integer> arr = new ArrayList<>();
      printNodes(root, arr, low, high);
      return arr;
    }
    static void printNodes(Node root, ArrayList<Integer> arr, int l, int h){
        if(root==null)
            return;
        if(root.data>=l)
            printNodes(root.left, arr, l, h);
        if(root.data>=l && root.data<=h)
            arr.add(root.data);
        if(root.data<=h)
            printNodes(root.right, arr, l, h);
    }

}
```

## 4. Pair Sum in BST

Given a BST and a number X. The task is to check if any pair exists in BST or not whose sum is equal to X.

One method is to use auxialiary array to store in-order traversal of BST. Then we can apply two pointer approach to find given sum.

Second one is,

1. Traverse tree inorder way to find if any pair exists which gives sum x.

2. Use hashset to keep check is pair exist or not. If not then add the root data to hashset.

```java
    static HashSet<Integer> hs;
    static boolean findPair(Node root, int sum) {
        // Your code
        hs = new HashSet<>();
        return find(root, sum);
    }
    static boolean find(Node root, int sum){
        if(root==null)
            return false;
        if(find(root.left, sum))
            return true;
        if(hs.contains(sum-root.data))
            return true;
        hs.add(root.data);
        return find(root.right, sum);
    }
```

## 5. Smaller on Right

You are given an integer array *nums* and you have to return a new *counts* array. The *counts* array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.

Every node will maintain a val, small_value recording the total of number on it's left bottom side, freq counts the frequency. For example, [3, 2, 2, 6, 1], from back to beginning, we would have:

```
1(0, 1)
     \
     6(3, 1)
     /
   2(0, 2)
       \
        3(0, 1)
```

When we try to insert a number, the total number of smaller numbers would be adding freq and small_values of the nodes where we turn right.

for example, if we insert 5, it should be inserted on the way down to the right of 3, the nodes where we turn right is 1(0,1), 2(0,2), 3(0,1), so the answer should be (0 + 1)+(0 + 2)+ (0 + 1) = 4

if we insert 7, the right-turning nodes are 1(0,1), 6(3,1), so answer should be (0 + 1) + (3 + 1) = 5

```
class Solution {
    int count;
    class TreeNode{
        int val, small_values, freq;
        TreeNode left, right;
        TreeNode(int val, int small_values){
            this.val = val;
            this.small_values = small_values;
            this.freq = 1;
            left=right=null;
        }
    }
    public List<Integer> countSmaller(int[] nums) {
        TreeNode root = null;
        Integer [] res = new Integer[nums.length];
        for(int i=nums.length-1;i>=0;i--){
            count = 0;
            root = insert(root, nums[i], 0);
            res[i] = count;
        }
        return Arrays.asList(res);
    }
    TreeNode insert(TreeNode root, int val, int sum){
        if(root==null){
            count = sum;
            return new TreeNode(val, 0);
        }
        if(root.val==val){
            root.freq++;
            count = sum + root.small_values;
        }
        else if(root.val>val){
            root.small_values++;
            root.left = insert(root.left, val, sum);
        }
        else
            root.right = insert(root.right, val, sum + root.small_values + root.freq);
        return root;
    }
}
```

## 6. Floor in BST

Given a Binary search tree and a value X, the task is to complete the function which will return the floor of x.

**Note:** Floor(X) is an element that is either equal to X or immediately smaller to X. If no such element exits return -1.

**Solution 1[Recursive]: Uses O(h) extra space and function call overhead**

```
int floor(Node root, int key)
{
    if(root==null)
        return -1;
    if(root.data>key)
        return floor(root.left, key);
    else
        return Math.max(floor(root.right, key), root.data);
}
```

**Solution 2[Iterative]: Uses O(1) extra space and O(h) time**

```
int floor(Node root, int key)
{
    Node ptr = root;
    int res = -1;
    while(ptr!=null){
        if(ptr.data>key)
            ptr = ptr.left;
        else{
            res = ptr.data;
            ptr = ptr.right;
        }
    }
    return res;
}
```

## 7. Ceil in BST

Given a BST and a number X. The task it to find Ceil of X.

**Note**: Ceil(X) is a number that is either equal to X or is immediately greater than X.

**Solution 1[Recursive]: Uses O(h) extra space and function call overhead**

```
int findCeil(Node root, int key) {
    if (root == null)
        return -1;
    if(root.data>=key){
        int left = findCeil(root.left, key);
        if(left!=-1)
            return left;
        return root.data;
    }
    return findCeil(root.right, key);
}
```

**Solution 2[Iterative]: Uses O (1) extra space and O(h) time**

```
int findCeil(Node root, int key) {
    if (root == null)
        return -1;
    int res = -1;
    while(root!=null){
        if(root.data>=key){
            res = root.data;
            root = root.left;
        }
        else
            root = root.right;
    }
    return res;
}
```

## 8. Find kth smallest in BST

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

**Example 1:**

```
Input: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
Output: 1
```

**Example 2:**

```
Input: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
Output: 3
```

**Solution 1 [DFS]: faster than solution 2**

```
class Solution {
    public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> st = new Stack<>();
        while(!st.isEmpty() || root!=null){
            if(root!=null){
                st.push(root);
                root = root.left;
            }
            else{
                root = st.pop();
                k--;
                if(k==0)
                    return root.val;
                root = root.right;
            }
        }
        return -1;
    }
}
```

Solution 2 [modify BST structure]: this solution is slower as compared to solution 1. But whenever there is requirement to frequent search for kthSmallest it gives in O(log n) time.

**Follow up:**

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Insert and delete in a BST were discussed last week, the time complexity of these operations is O(H), where H is a height of binary tree, and H = logN for the balanced tree.

Hence without any optimisation insert/delete + search of kth element has $O(2H+k)$ complexity. How to optimise that?

That's a design question, basically we're asked to implement a structure which contains a BST inside and optimises the following operations:

Insert

Delete

Find kth smallest

Seems like a database description, isn't it? Let's use here the same logic as for LRU cache design, and combine an indexing structure (we could keep BST here) with a double linked list.

Such a structure would provide:

$O(H)$ time for the insert and delete.

O($k$) for the search of kth smallest.

The idea is to maintain class member lcount which keeps track of number node on left subtree.

**Solution 2:**

```
class Solution {
    class Node{
        Node left, right;
        int val, count;
        Node(int val){
            this.val = val;
        }
    }
    public int kthSmallest(TreeNode root, int k) {
        Node root_node = insert(root);
        return kthSmallest(root_node, k);
    }
    Node insert(TreeNode root){
        if(root==null)
            return null;
        Node root_node = new Node(root.val);
        root_node.left = insert(root.left);
        root_node.right = insert(root.right);
        if(root_node.left!=null)
            root_node.count = countSmaller(root_node.left);
        return root_node;
    }
    int countSmaller(Node root_node){
        if(root_node==null)
            return 0;
        return 1 + countSmaller(root_node.left) + countSmaller(root_node.right);
    }
    int kthSmallest(Node root_node, int k){
        if(root_node.count+1 == k)
            return root_node.val;
        if(root_node.count+1 > k)
            return kthSmallest(root_node.left, k);
        return kthSmallest(root_node.right, k-root_node.count-1);
    }
}
```

## 9. Preorder to postorder

Given an array arr[] of N nodes representing preorder traversal of BST. The task is to print its postorder traversal.

```
Input:
N = 5
arr[]  = {40,30,35,80,100}
Output: 35 30 100 80 40
Explanation: PreOrder: 40 30 35 80 100
InOrder: 30 35 40 80 100
Therefore, the BST will be:
              40
           /      \
         30        80
           \         \
           35        100
Hence, the postOrder traversal will
be: 35 30 100 80 40
```

```
static int index;
public static Node constructTree(int pre[], int size) {
    //Your code here
    index = 0;
    return construct(pre, Integer.MIN_VALUE, Integer.MAX_VALUE, size);
}
static Node construct(int arr[], int lower, int upper, int n){
    if(index<n && arr[index]>lower && arr[index]<upper){
        Node root = new Node(arr[index++]);
        root.left = construct(arr, lower, root.data, n);
        root.right = construct(arr, root.data, upper, n);
        return root;
    }
    return null;
}
```

### 10. Merge two BST 's

Given two BST, Return elements of both BSTs in sorted form.

```
BST1:
       5
     /   \
    3     6
   / \
  2   4
BST2:
        2
      /   \
     1     3
            \
             7
            /
           6
BST2 = 2 1 3 N N N 7 6
Output: 1 2 2 3 3 4 5 6 6 7
Explanation: After merging and sorting the
two BST we get 1 2 2 3 3 4 5 6 6 7
```

```
public List<Integer> merge(Node root1,Node root2)
{
    Stack<Node> st1 = new Stack<>();
    Stack<Node> st2 = new Stack<>();
    ArrayList<Integer> res = new ArrayList<>();
    while(!st1.isEmpty() || !st2.isEmpty() || root1!=null || root2!=null){
        if(root1!=null || root2!=null){
            if(root1!=null){
                st1.push(root1);
                root1 = root1.left;
            }
            if(root2!=null){
                st2.push(root2);
                root2 = root2.left;
            }
        }
        else{
            if(!st1.isEmpty())
                root1 = st1.pop();
            if(!st2.isEmpty())
                root2 = st2.pop();
            if(root1!=null && root2!=null){
                if(root1.data<root2.data){
                    res.add(root1.data);
                    root1 = root1.right;
                    st2.push(root2);
                    root2 = null;
                }
                else{
                    res.add(root2.data);
                    root2 = root2.right;
                    st1.push(root1);
                    root1 = null;
                }
            }
            else if(root1==null){
                res.add(root2.data);
                root2 = root2.right;
            }
            else if(root2==null){
                res.add(root1.data);
                root1 = root1.right;
            }
        }
    }
    return res;
}
```

## 11. Fixing Two nodes of a BST

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST by swapping them back. Do not change the structure of the tree.

Note: It is guaranteed than the given input will form BST, except for 2 nodes that will be wrong.

**Solution:**

The idea is similar to finding two elements from sorted array which are miss-ordered. Given an sorted array which contains two element as misplaced.

[4, 60, 10, 20, 8, 80, 100] → [60, 8] are swapped

[4, 8, 10, 60, 20, 80, 100] → [60, 20] are swapped

```
int prev = arr[0], first=null, second=null;
for(int i=1;i<n;i++){
    if(arr[i] <= prev){
        if(first==null)
          first = prev;
        second = arr[i];
    }
    prev = arr[i];
}
```

The given problem reduces to this if we do in-order traversal of given BST. Inorder traversal will give us sorted list. On this sorted list we can apply above approach. But in below approach we don't need to maintain arraylist.

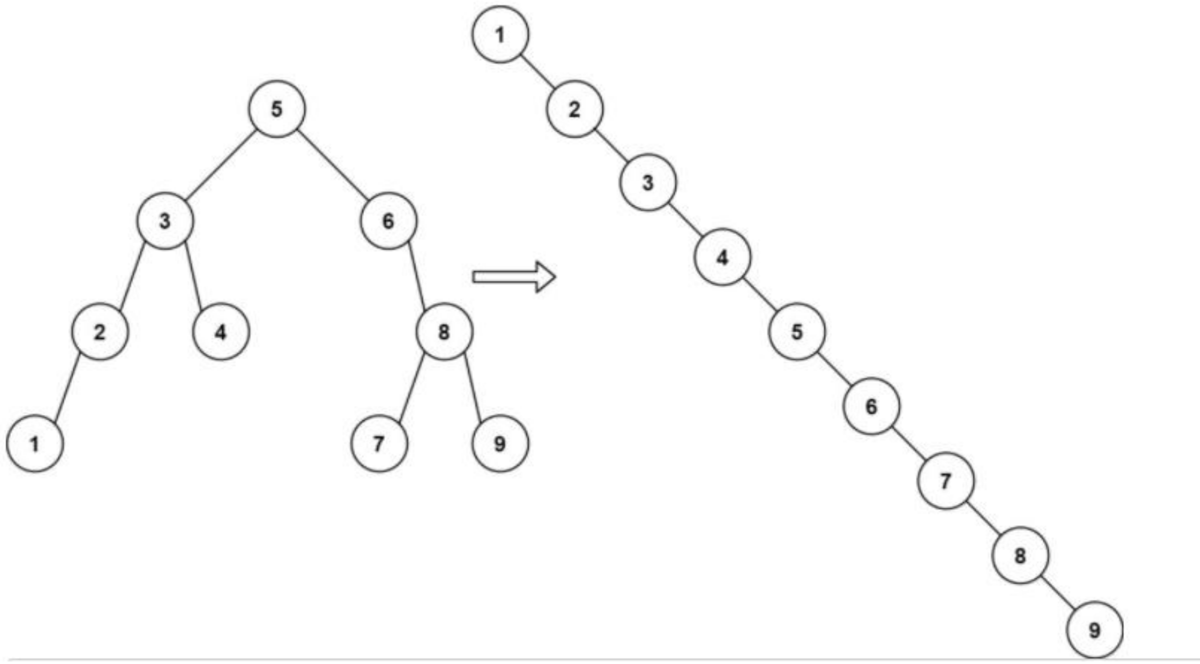
```
class Sol
{
    Node prev, first, second;
    public Node correctBST(Node root)
    {
        //code here.
        first = second = prev = null;
        findSwapped(root);
        int temp = first.data;
        first.data = second.data;
        second.data = temp;
        return root;
    }
    void findSwapped(Node root){
        if(root==null)
            return;
        findSwapped(root.left);
        if(prev!=null && root.data<=prev.data){
            if(first==null)
                first = prev;
            second = root;
        }
        prev = root;
        findSwapped(root.right);
    }
}
```

## Self-Balancing BST

In normal BST we can perform all the operations (search, insert, delete, floor, ceil, smaller and greater) in Big O(h) time. But if tree is not balanced sometimes it will take O(n) time.

Where, h: height of the BST, and n: number of nodes in BST

Suppose, we are inserting (stream of) data in BST in ascending/descending order we will get left/right skewed BST. In such BST it will take O(n) time search data.

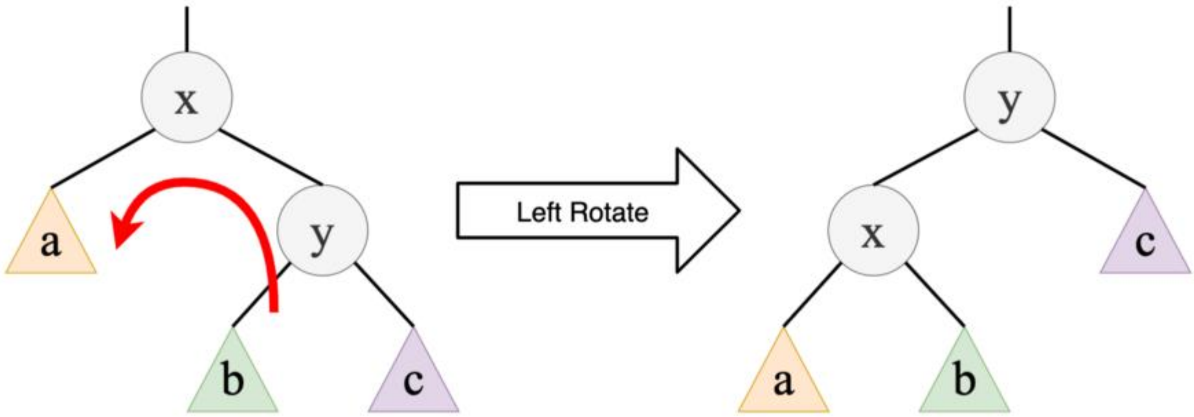



**Self-Balancing Binary Search Trees** are **height-balanced** binary search trees that automatically keeps height as small as possible when insertion and deletion operations are performed on tree. The height is typically maintained in order of $\log n$ so that all operations take θ ($\log n$) time on average.

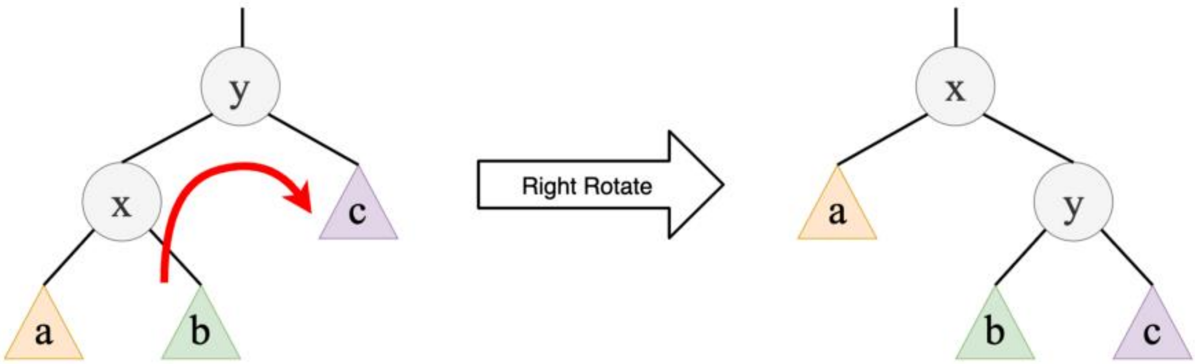
### How do Self-Balancing BSTs Balance?

When it comes to self-balancing, BSTs perform rotations after performing insert and delete operations. Given below are the two types of rotation operations that can be performed to balance BSTs without violating the binary-search-tree property.

1. **Left rotation**
   Left rotation on node x




2. **Right rotation**
   Right rotation on node x




## Types of Self-Balancing BSTs

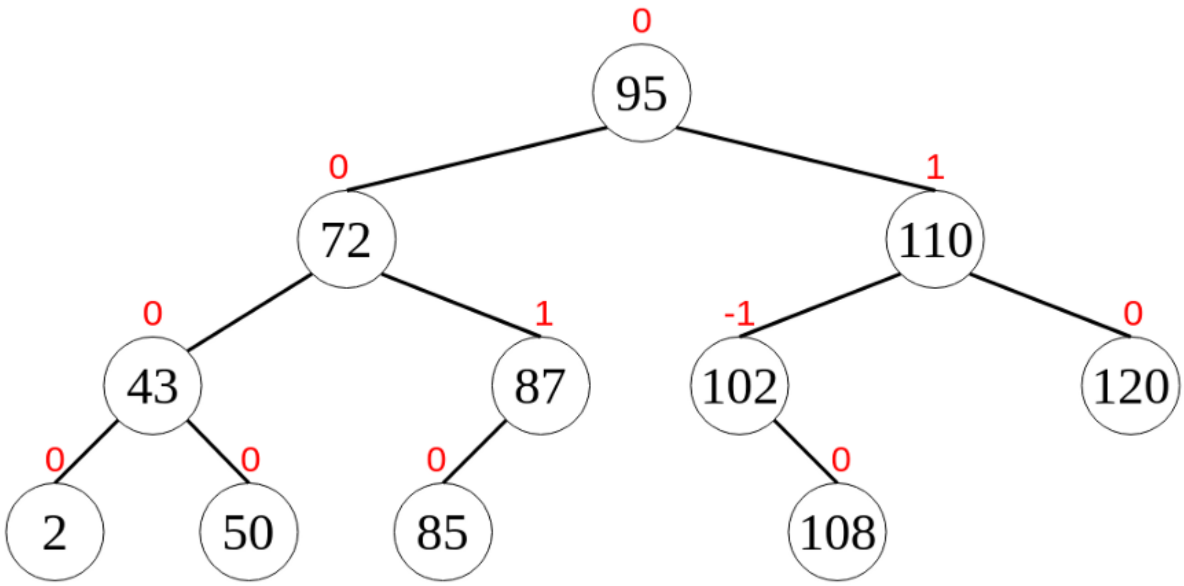
- AVL trees
- Red-black trees
- Splay trees
- Treaps

## AVL Trees

All the node in an AVL tree stores their own balance factor.

In an AVL tree, the balance factor of every node is either -1, 0 or +1. In other words, the difference between the height of the left subtree and the height of the right subtree cannot be more than 1 for all of the nodes in an AVL tree.

In Figure, the values in red colour above the nodes are their corresponding balance factors. You can see that the balance factor condition is satisfied in all the nodes of the AVL tree shown in Figure.




## Rotations in AVL Trees

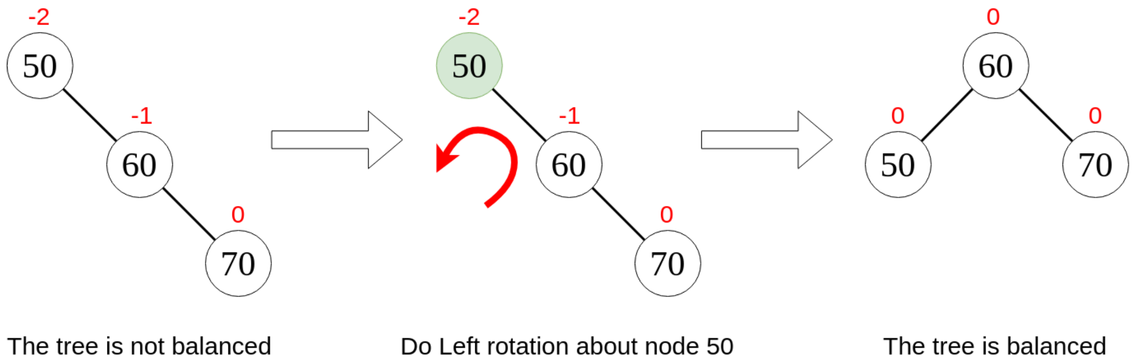
After performing insertions or deletions in an AVL tree, we have to check whether the balance factor condition is satisfied by all the nodes. If the tree is not balanced, then we have to do rotations to make it balanced.

Rotations performed on AVL trees can be of four main types that are grouped under two categories. They are,

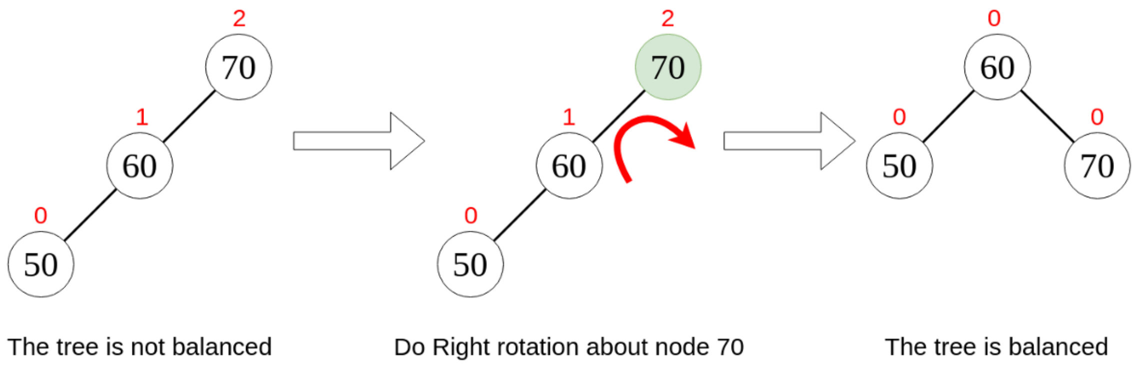
**Single rotations — Left (LL) Rotation and Right (RR) Rotation**

**Double rotations — Left Right (LR) Rotation and Right Left (RL) Rotation**

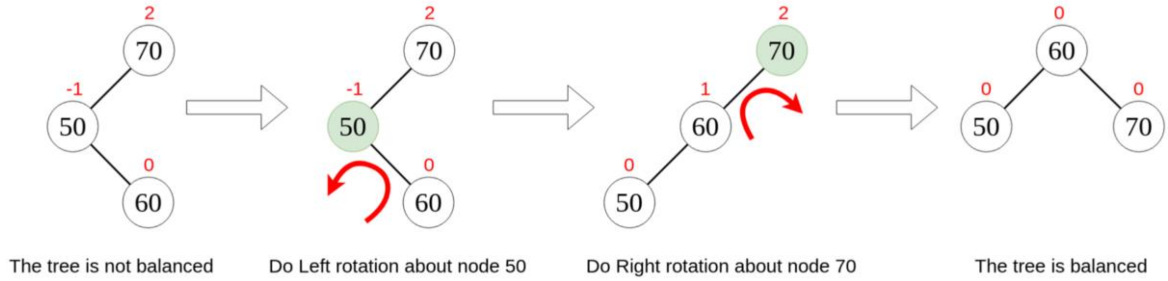
1. **Single Left Rotation (LL Rotation)**



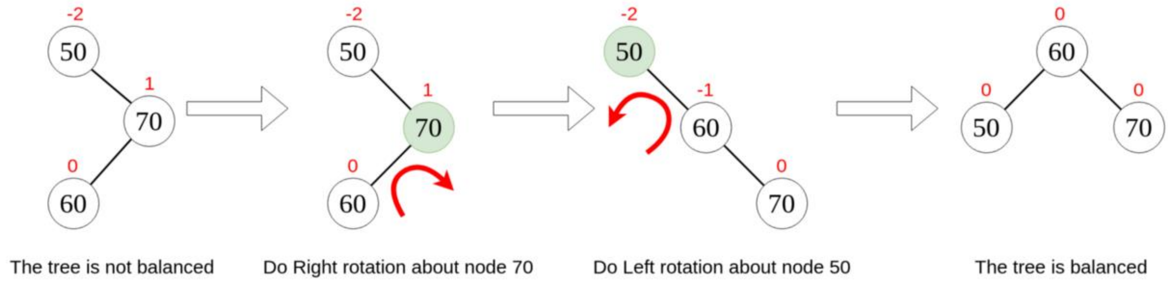

2. **Single Right Rotation (RR Rotation)**




3. **Left Right Rotation (LR Rotation)**

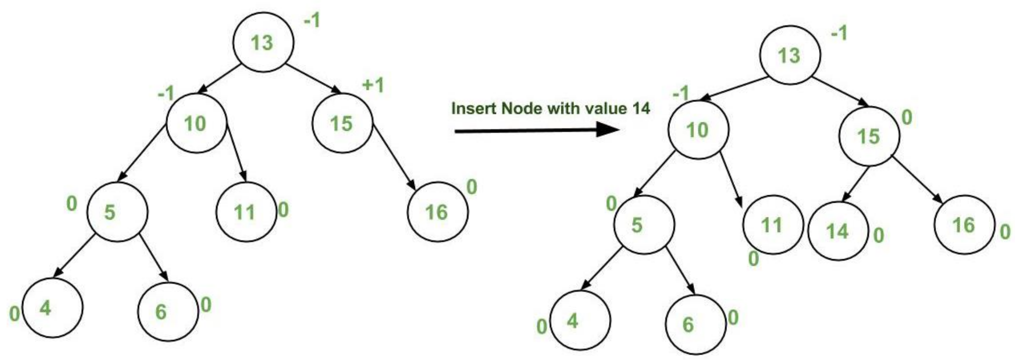



4. **Right Left Rotation (RL Rotation)**

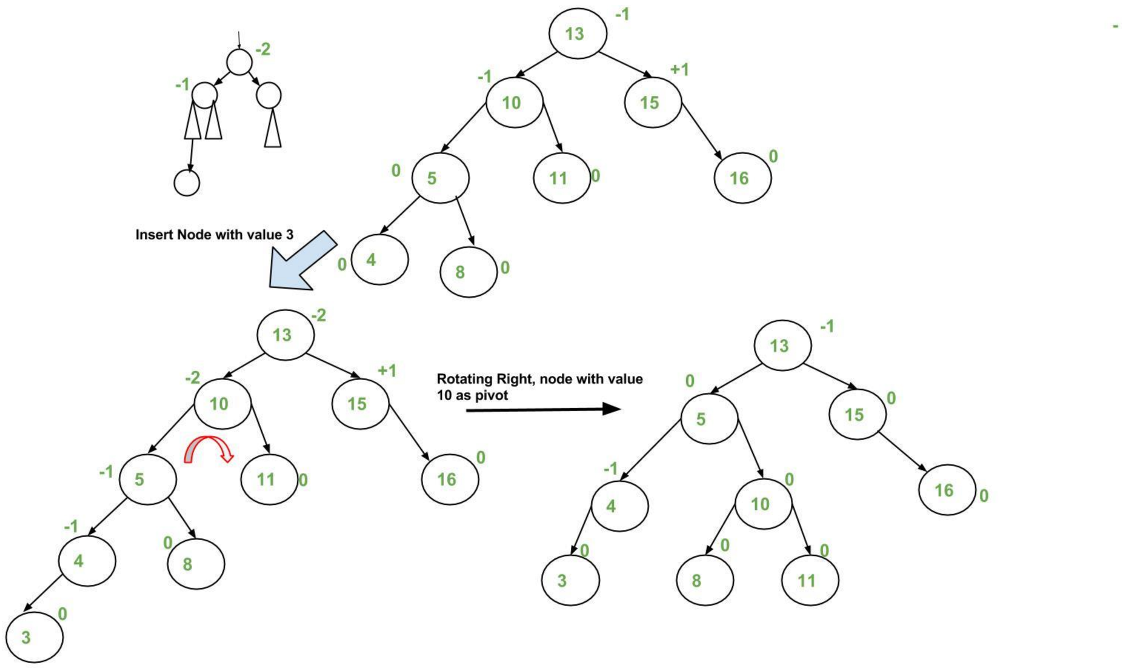



## Insertion examples

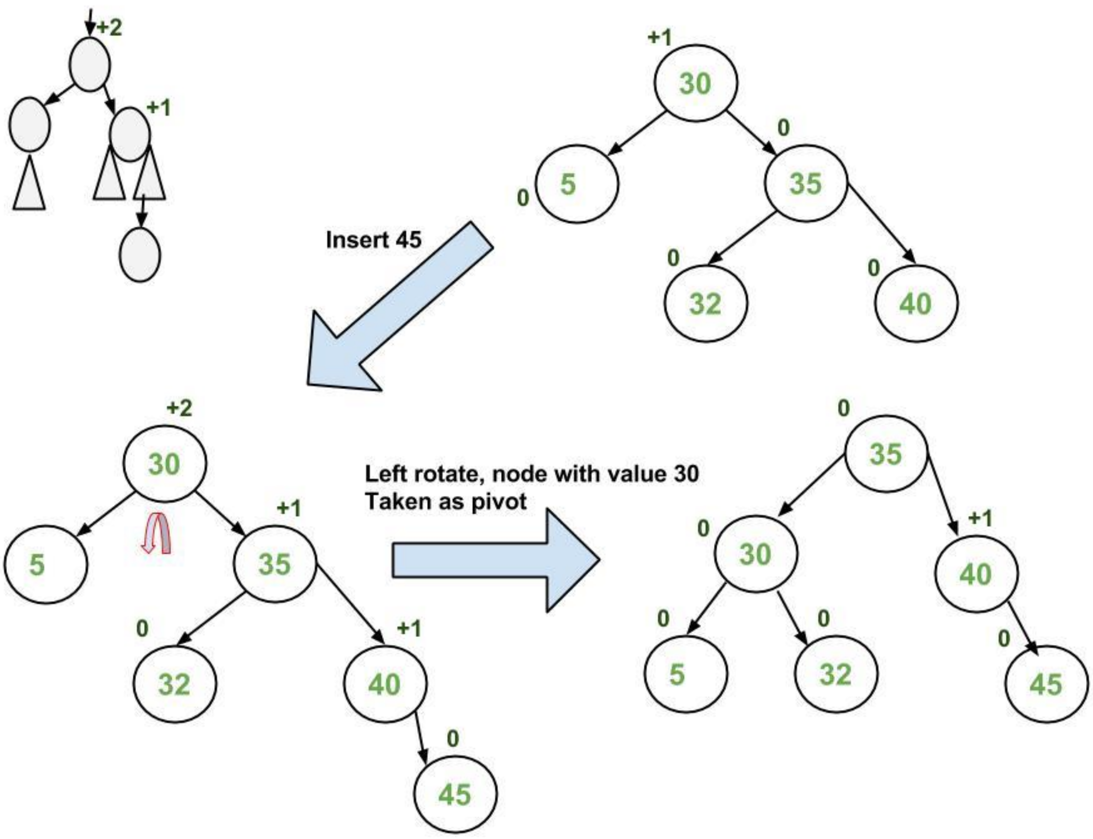
1. **No need to rotate BST**



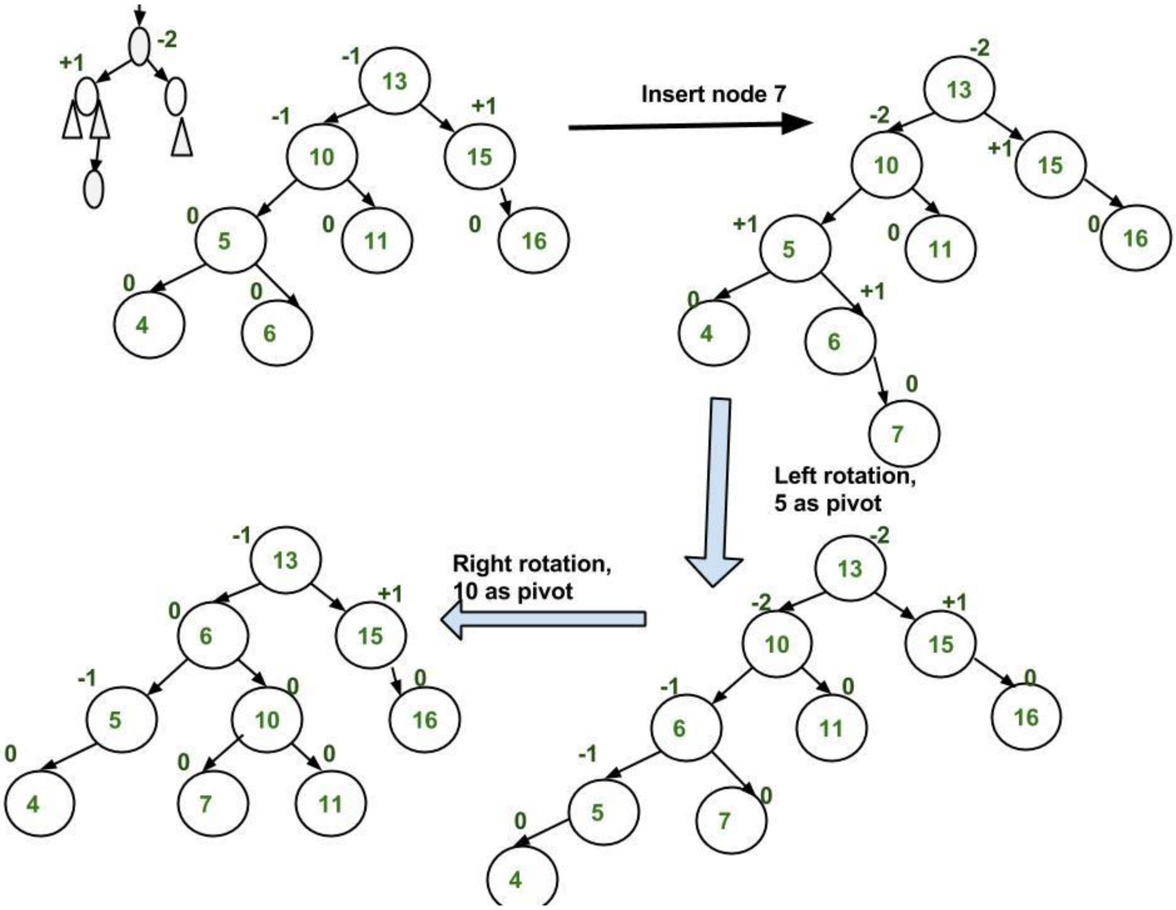
## 2. Right rotate



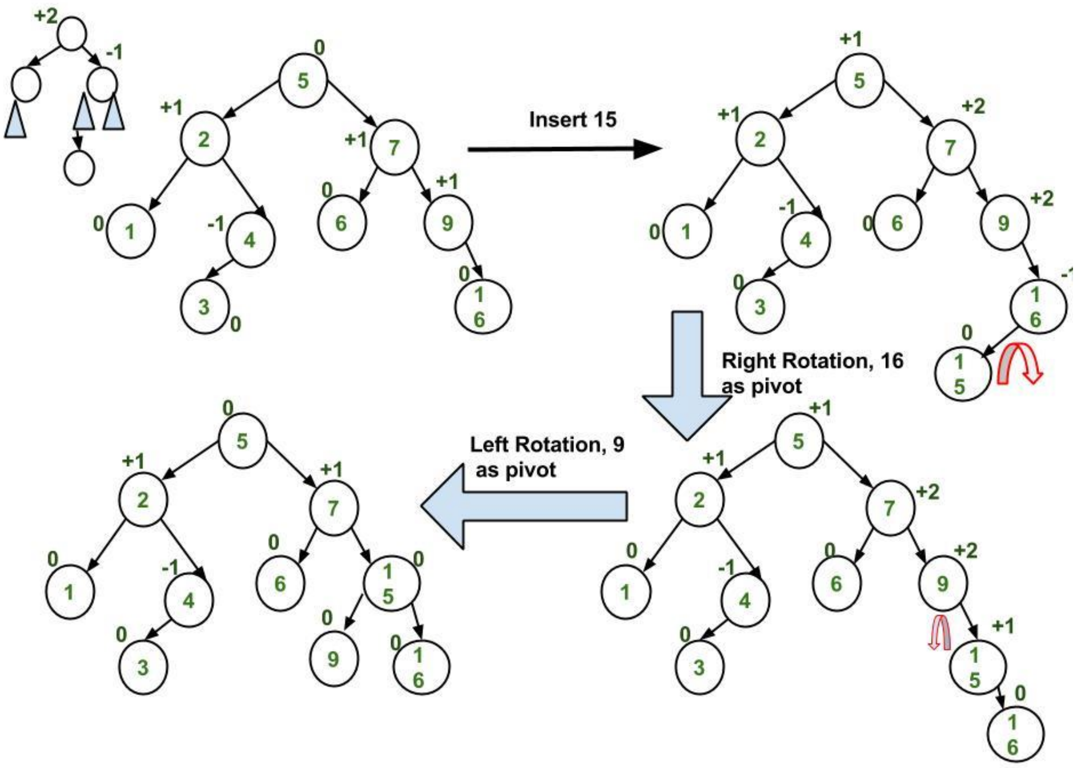

## 3. Left rotate



## 4. Left Right rotate




## 5. Right rotate



It seems that we are doing extra work in every insertion. But if we look closer, we'll found out that in every insertion we only need to check valance factor all the ancestors of newly inserted node. And we can do this in O ($\log n$), if find any of the ancestor with invalid balance factor we do rotation accordingly. And this rotation is constant work. Furthermore, to find balance factor of node we'll require height of left-subtree and right-subtree, and finding height of any binary tree is also O ($\log n$) work.

## Red-Black Tree

### Introduction

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black). These colours are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, but is good enough to reduce the searching time and maintain it around O (log n) time, where n is the total number of elements in the tree.

### Rules for Red-Black Tree

1. Every node has a colour either red or black.
2. The root of tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendant NULL node (leaf node) has the same number of black nodes.

### Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete, ... etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O (log n) after every insertion and deletion, then we can guarantee an upper bound of O (log n) for all these operations. The height of a Red-Black tree is always O (log n) where n is the number of nodes in the tree.

### Comparison with AVL

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So, if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

### Black Height of a Red-Black Tree

Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes. From the above properties 3 and 4, we can derive, a Red-Black Tree of **height h has black-height >= h/2.**

*Number of nodes from a node to its farthest descendant leaf is no more than twice as the number of nodes to the nearest descendant leaf.*

### Applications of Self-Balancing Binary Search Trees

**To maintain stream of data** in sorted order (stream of data coming in sorted order but not necessary to be sorted).

**To implement doubly ended priority queue.** Singly ended priority queue can be implemented by Heap data structure. Singly ended priority queues gives either maximum or minimum in O(1) time. Self-balancing BSTs provide both maximum and minimum in O(1) time.

To solve problems like:

Count smaller/greater in stream

Find floor, ceil, greater, smaller, etc, … in a stream

## Problems

### 1. Check for BST

Given a binary tree. Check whether it is a BST or not.

```
public class Tree{
    boolean isBST(Node root){
        return isBST(root, null, null);
    }
    boolean isBST(Node root, Integer lower, Integer upper){
        if(root==null)
            return true;
        if(lower!=null && root.data<=lower)
            return false;
        if(upper!=null && root.data>=upper)
            return false;
        return isBST(root.left, lower, root.data) && isBST(root.right, root.data, upper);
    }
}
```

Another approach is to use in-order traversal of BST.

In-order traversal of BST gives sorted list of elements. There are two solution to use in-order traversal: One thing we can do is do in-order traversal of BST and check whether the returned list of elements is sorted or not. Second way is to maintain global variable prev.

```
class Solution {
    Integer prev = null;
    public boolean isValidBST(TreeNode root) {
        if(root==null)
            return true;
        if(!isValidBST(root.left))
            return false;
        if(prev!=null && root.val<=prev)
            return false;
        prev = root.val;
        return isValidBST(root.right);
    }
}
```

## 2. Find closest element in BST

Given a BST and an integer. Find the least absolute difference between any node value of the BST and the given integer.

```
Input:
          10
        /    \
       2      11
     /   \
    1     5
        /   \
       3     6
        \
         4
K = 13
Output: 2
Explanation: K=13. The node that has
value nearest to K is 11. so the answer
is 2
```

```
class Solution{
    static int res;
    static int maxDiff(Node  root, int K) {
        res = Integer.MAX_VALUE;
        maxDiffUtil(root, K);
        return res;
    }
    static void maxDiffUtil(Node root, int k){
        if(root==null)
            return;
        res = Math.min(res, Math.abs(k-root.data));
        if(root.data>k)
            maxDiffUtil(root.left, k);
        else
            maxDiffUtil(root.right, k);
    }
}
```

## 3. Convert level-order traversal to BST

Given an array of size N containing level order traversal of a BST. The task is to complete the function constructBst(), that construct the BST (Binary Search Tree) from its given level order traversal.

**Solution 1[Recursive]: Uses O(h) extra space (function call overhead) and O $(n\log n)$ time**

```java
    public Node constructBST(int[] arr){
        Node root = null;
        for(int i=0;i<arr.length;i++)
            root = insert(root, arr[i]);
        return root;
    }
    Node insert(Node root, int val){
        if(root==null)
            return new Node(val);
        if(root.data>val)
            root.left = insert(root.left, val);
        else
            root.right = insert(root.right, val);
        return root;
    }
```

**Solution 2[BFS]: Uses O(N) extra space (maintain queue) and O (n) time**

```java
class GFG {
    class NodeLimit{
        Node node;
        int lower, upper;
        NodeLimit(Node node, int lower, int upper){
            this.node = node;
            this.lower = lower;
            this.upper = upper;
        }
    }
    public Node constructBST(int[] arr){
        //Write your code here and return the root of the constructed BST
        Node root = new Node(arr[0]);
        Queue<NodeLimit> q = new LinkedList<>();
        q.add(new NodeLimit(root, Integer.MIN_VALUE, Integer.MAX_VALUE));
        for(int i=1;i<arr.length;){
            NodeLimit curr = q.poll();
            if(arr[i]>curr.lower && arr[i]<curr.node.data){
                curr.node.left = new Node(arr[i++]);
                q.add(new NodeLimit(curr.node.left, curr.lower, curr.node.data));
            }
            if(i<arr.length && arr[i]>curr.node.data && arr[i]<curr.upper){
                curr.node.right = new Node(arr[i++]);
                q.add(new NodeLimit(curr.node.right, curr.node.data, curr.upper));
            }
        }
        return root;
    }
}
```

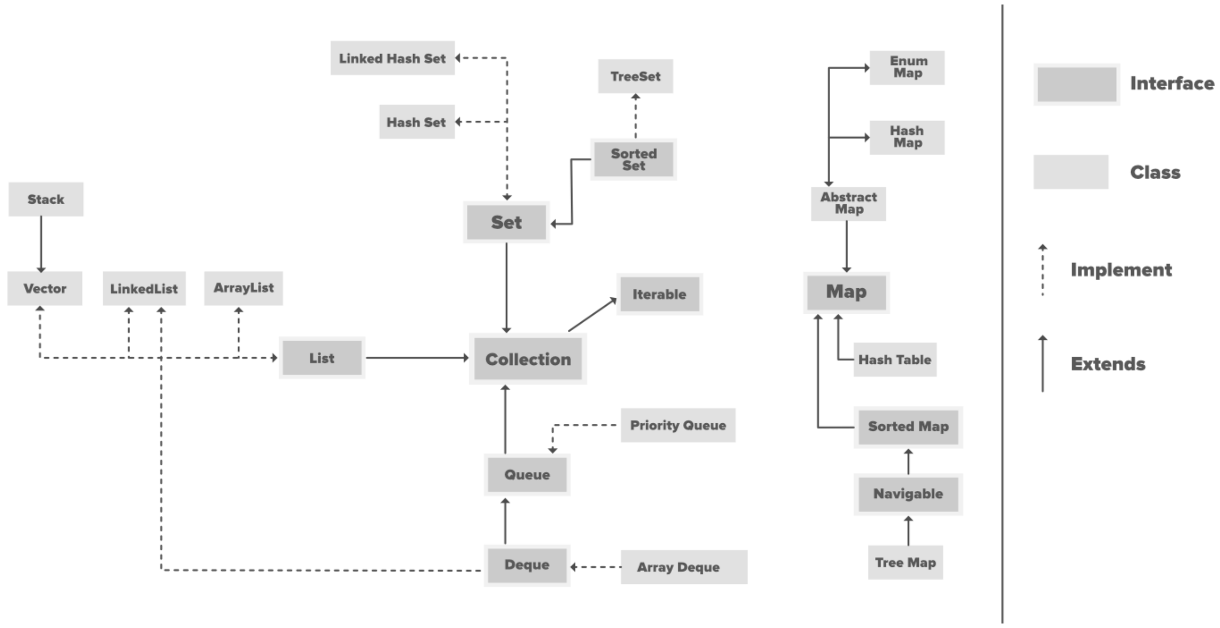
### 4. Count BST nodes that lie in the given range

Given a Binary Search Tree (BST) and a range l-h(inclusive), count the number of nodes in the BST that lie in the given range.

```
Input:
      10
     /  \
    5    50
   /    /  \
  1    40  100
l = 5, h = 45
Output: 3
Explanation: 5 10 40 are the node in the
range
```

```java
    static int res;
    public static int getCountOfNode(Node root,int l, int h)
    {
        res = 0;
        helper(root, l, h);
        return res;
    }
    static void helper(Node root, int l, int h){
        if(root==null)
            return;
        if(root.data>=l && root.data<=h)
            res++;
        if(root.data>l)
            helper(root.left, l, h);
        if(root.data<h)
            helper(root.right, l, h);
    }
```

# TreeSet and TreeMap in Java




TreeSet and TreeMap both are self-balancing BSTs, implemented using red-black binary tree. It is similar to HashSet and HashMap.

It provides functionalities that are similar to HashSet/HashMap but in additional functionality it provides finding floor, ceiling, higher, and lower.

Similar to HashSet and HashMap TreeSet and TreeMap does not maintain inertion order but instead it **inserts element in sorted order (ascending)**.

Since it is implementation of self-balancing binary trees it performs operations like **insert, deleted, search, floor, ceiling, higher, lower in O ($\log n$)** time.

TreeSet maintains only keys. While TreeMap gives functionalities to store value corresponding to key (key-value pair).

## Problems on TreeSet/TreeMap

1. Vertical Traversal of binary tree

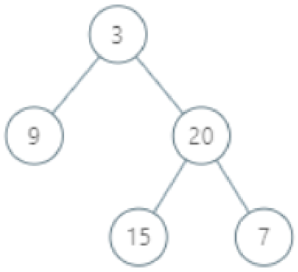
Given a binary tree, return the *vertical order* traversal of its nodes values.

For each node at position `(X, Y)`, its left and right children respectively will be at positions `(X-1, Y-1)` and `(X+1, Y-1)`.

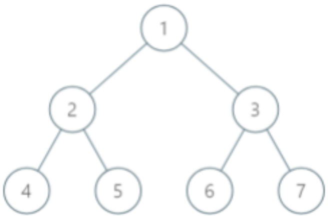
Running a vertical line from `X = -infinity` to `X = +infinity`, whenever the vertical line touches some nodes, we report the values of the nodes in order from top to bottom (decreasing `Y` coordinates).

If two nodes have the same position, then the value of the node that is reported first is the value that is smaller.

Return an list of non-empty reports in order of `X` coordinate. Every report will have a list of values of nodes.




```
Input: [3,9,20,null,null,15,7]
Output: [[9],[3,15],[20],[7]]
Explanation:
Without loss of generality, we can assume the root node is at position (0, 0):
Then, the node with value 9 occurs at position (-1, -1);
The nodes with values 3 and 15 occur at positions (0, 0) and (0, -2);
The node with value 20 occurs at position (1, -1);
The node with value 7 occurs at position (2, -2).
```




```
Input: [1,2,3,4,5,6,7]
Output: [[4],[2],[1,5,6],[3],[7]]
Explanation:
The node with value 5 and the node with value 6 have the same position according to the given
scheme.
However, in the report "[1,5,6]", the node value of 5 comes first since 5 is smaller than 6.
```

**Solution 1 [DFS]:**

Time complexity: $O(n + n\log n + n + n)$ --> O(insert in arraylist + sort arraylist + insert in hashmap from sorted list + insert in resultant list from hashmap)

Space complexity: $O(n + h + n + n)$ --> O(arraylist + recursion call stack + hashmap + resultant arraylist)

```
class Solution {
    class Point{
        int val, x, y;
        Point(int val, int x, int y){
            this.val = val;
            this.x = x;
            this.y = y;
        }
    }
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        if(root==null)
            return new ArrayList<List<Integer>>();
        ArrayList<Point> points = new ArrayList<>();
        insert(root, 0, 1, points);
        Collections.sort(points, (a, b) -> a.x!=b.x ? a.x-b.x : a.y!=b.y ? a.y-b.y : a.val-b.val);
        LinkedHashMap<Integer, List<Integer>> hm = new LinkedHashMap<>();
        for(Point p : points){
            List<Integer> list = hm.getOrDefault(p.x, new ArrayList<>());
            list.add(p.val);
            hm.put(p.x, list);
        }
        ArrayList<List<Integer>> res = new ArrayList<>();
        for(Map.Entry it : hm.entrySet())
            res.add((List<Integer>)it.getValue());
        return res;
    }
    void insert(TreeNode root, int x, int level, ArrayList<Point> points){
        if(root==null)
            return;
        points.add(new Point(root.val, x, level));
        insert(root.left, x-1, level+1, points);
        insert(root.right, x+1, level+1, points);
    }
}
```

**Solution 2 [BFS]: Uses TreeMap**

Time complexity: $O(n\log n + n)$ --> O(tree traversal with accessing treemap + insert into resultant list from treemap)

Inserting/searching from treemap takes $O(\log n)$ time.

Space complexity: $O(n + n + n)$

```java
class BinaryTree
{
    static class NodeX{
        Node node;
        int x;
        NodeX(Node node, int x){
            this.node = node;
            this.x = x;
        }
    }
    static ArrayList <Integer> verticalOrder(Node root){
        if(root==null)
            return new ArrayList<>();
        Queue<NodeX> q = new LinkedList<>();
        TreeMap<Integer, ArrayList<Integer>> tm = new TreeMap<>();
        q.add(new NodeX(root, 0));
        while(!q.isEmpty()){
            NodeX curr = q.remove();
            if(curr.node.left!=null)
                q.add(new NodeX(curr.node.left, curr.x-1));
            if(curr.node.right!=null)
                q.add(new NodeX(curr.node.right, curr.x+1));
            ArrayList<Integer> list;
            if(!tm.containsKey(curr.x))
                list = new ArrayList<>();
            else
                list = tm.get(curr.x);
            list.add(curr.node.data);
            tm.put(curr.x, list);
        }

        ArrayList<Integer> res = new ArrayList<>();
        for(Map.Entry it : tm.entrySet()){
            ArrayList<Integer> list = (ArrayList<Integer>) it.getValue();
            for(int val : list)
                res.add(val);
        }
        return res;
    }
}
```

### 2. Top view of Binary Tree

Given below is a binary tree. The task is to print the top view of binary tree. Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. For the given below tree.

**Solution 1[using HashMap]: extra space: function call overhead and O ($n + n\log n + n$) time (insert in arraylist + to sort according to position of node + traverse hashmap).**

```java
class View{
    static class Point{
        int val, x, y;
        Point(int val, int x, int y){
            this.val = val;
            this.x = x;
            this.y = y;
        }
    }
    static void topView(Node root){
        if(root==null)
            return;
        ArrayList<Point> points = new ArrayList<>();
        insert(root, 0, 1, points);
        Collections.sort(points, (a, b) -> a.x!=b.x ? a.x-b.x : a.y!=b.y ? a.y-b.y : a.val-b.val);

        HashSet<Integer> hs = new HashSet<>();
        for(Point p : points){
            if(!hs.contains(p.x))
                System.out.print(p.val+" ");
            hs.add(p.x);
        }
    }
    static void insert(Node root, int x, int level, ArrayList<Point> points){
        if(root==null)
            return;
        points.add(new Point(root.data, x, level));
        insert(root.left, x-1, level+1, points);
        insert(root.right, x+1, level+1, points);
    }
}
```

**Solution 2[using TreeMap]: extra space: Queue to do BFS and O ($n \log n + n$) time (traversing tree and add it to queue + traversing treemap).**

```java
class View{
    static class NodeX{
        Node node;
        int x;
        NodeX(Node node, int x){
            this.node = node;
            this.x = x;
        }
    }
    static void topView(Node root){
        if(root==null)
            return;
        Queue<NodeX> q = new LinkedList<>();
        q.add(new NodeX(root, 0));
        TreeMap<Integer, Integer> tm = new TreeMap<>();
        while(!q.isEmpty()){
            NodeX curr = q.remove();
            if(curr.node.left!=null)
                q.add(new NodeX(curr.node.left, curr.x-1));
            if(curr.node.right!=null)
                q.add(new NodeX(curr.node.right, curr.x+1));
            if(!tm.containsKey(curr.x))
                tm.put(curr.x, curr.node.data);
        }
        for(Map.Entry it : tm.entrySet())
            System.out.print(it.getValue() + " ");
    }
}
```

### 3. Bottom view of Binary Tree

Given a binary tree, print the bottom view from left to right.

```
      20
     /  \
    8    22
   / \     \
  5   3     25
     / \
    10  14
```

For the above tree, the bottom view is 5 10 3 14 25.

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottommost nodes at horizontal distance 0, we need to print 4.

```
        20
       /  \
      8    22
     / \   / \
    5   3 4   25
       / \
     10   14
```

For the above tree the output should be 5 10 4 14 25.

```java
class NodeX{
    Node node;
    int x;
    NodeX(Node node, int x){
        this.node = node;
        this.x = x;
    }
}
public ArrayList <Integer> bottomView(Node root){
    if(root==null)
        return new ArrayList<>();
    ArrayList<Integer> res = new ArrayList<>();
    TreeMap<Integer, Integer> tm = new TreeMap<>();
    Queue<NodeX> q = new LinkedList<>();
    q.add(new NodeX(root, 0));
    while(!q.isEmpty()){
        NodeX curr = q.remove();
        if(curr.node.left!=null)
            q.add(new NodeX(curr.node.left, curr.x-1));
        if(curr.node.right!=null)
            q.add(new NodeX(curr.node.right, curr.x+1));
        if(tm.containsKey(curr.x))
            tm.remove(curr.x);
        tm.put(curr.x, curr.node.data);
    }
    for(Map.Entry<Integer, Integer> it : tm.entrySet())
        res.add((int)it.getValue());
    return res;
}
```

## 4. Ceiling on left side in an Array

Given array of integers we need to find ceiling of every element from left side.

(Ceiling means element that are smallest greater than equal to itself)

We did similar problem named previous greater which was implement using stack. But this problem different here we are asked to find ceiling of element (smallest greater element).

*Input : arr[] = {10, 5, 11, 6, 20, 12}*

*Output : -1, 10, -1, 10, -1, 20*

*First element has nothing on left side, so answer for first is -1.*

*Second element 5 has 10 on left, so the answer is 10.*

*Third element 11 has nothing greater or the same, so the answer is -1.*

*Fourth element 6 has 10 as value wise closes, so the answer is 10*

*Similarly we get values for fifth and sixth elements.*

*Input : arr[] = {10, 5, 11, 10, 20, 12}*

*Output : -1, 10, -1, 10, -1, 20*

```
static void printPrevGreater(int[] arr, int n)
{
    TreeSet<Integer> ts = new TreeSet<>();
    for (int i = 0; i < n; i++) {
        Integer c = ts.ceiling(arr[i]);
        if (c == null) // If no greater found
            System.out.print(-1 + " ");
        else
            System.out.print(c + " ");

        // Then insert
        ts.add(arr[i]);
    }
}
```

### 5. Vertical sum

Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines starting from left-most vertical line to right-most vertical line.

```
        1
      /   \
    2       3
   / \     / \
  4   5   6   7
```

The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4

Vertical-Line-2: has only one node 2=> vertical sum is 2

Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is 1+5+6 = 12

Vertical-Line-4: has only one node 3 => vertical sum is 3

Vertical-Line-5: has only one node 7 => vertical sum is 7

So expected output is 4, 2, 12, 3 and 7

```java
class Tree{
    public ArrayList <Integer> verticalSum(Node root) {
        TreeMap<Integer, Integer> tm = new TreeMap<>();
        verticalSum(root, 0, tm);
        ArrayList<Integer> res = new ArrayList<>();
        for(Map.Entry it : tm.entrySet())
            res.add((int)it.getValue());
        return res;
    }
    void verticalSum(Node root, int x, TreeMap<Integer, Integer> tm){
        if(root==null)
            return;
        int sum = tm.getOrDefault(x, 0);
        tm.put(x, sum+root.data);
        verticalSum(root.left, x-1, tm);
        verticalSum(root.right, x+1, tm);
    }
}
```